CSEP 505: Programming Languages

Lecture 6

February 12, 2015

```
desugar :: Expr → Result CExpr
desugar (WithStarE bindings body) =
  case bindings of
    [] → desugar body
    ((var, exp):bs) →
      case desugar (WithStarE bs body) of
        Ok b →
          Ok (AppC (FunC var b) (desugar exp))
        Err msg → Err msg
```

```
desugar (AppE exprs) = case exprs of
   [] → Err “empty app”
   [e] → Err “app with only one sub-expression”
   [f, a] → case (desugar f, desugar a) of
     (Ok f′, Ok a′) → Ok (AppC f′ a′)
   (f:e:es) → desugar (AppE (AppE f e):es)
desugar (FunE vars body) = case vars of
   [] → Err “no-arg function”
   [var] → FunC var (desugar body)
   (v:vs) → FunC v (desugar (FunE vs body))
```

```
interp :: Expr → Env → Result Val
interp (AppE fun arg) env =
  do fv ← interp fun env
     av ← interp arg env
     case fv of
       FunV var body closEnv → interp body ((var, av):closEnv)
       PrimV fn → fn av
       nonFun → fail …
interp (IfE cond cons alt) env =
  do cv ← interp cond env
     case cv of
       BoolV True → interp cons env
       BoolV False → interp alt env
       nonBool → fail …
```

```
instance Monad STR where
  return v = STR (\s → (Ok v, s))
  st >>= f = STR (\s → case st s of
    (Ok v, s′) → let (STR st′) = f v in
                 st′ s′
    (Err msg, s′) → (Err msg, s′))
```

```
interp :: Expr → Env → STR Val
interp (AppE fun arg) env =
  do fv ← interp fun env
     av ← interp arg env
     case fv of
       FunV var body closEnv → interp body ((var, av):closEnv)
       PrimV fn → fn av
       nonFun → fail …
interp (IfE cond cons alt) env =
  do cv ← interp cond env
     case cv of
       BoolV True → interp cons env
       BoolV False → interp alt env
       nonBool → fail …
```

```
type Stack a = [a]
push x stack = x:stack
peek (x:_) = x
pop (_:stack) = stack

type Queue a = [a]
enqueue :: a -> Queue a -> Queue a
enqueue x q = q ++ [x]
dequeue :: Queue a -> (a, Queue a)
dequeue (x:q) = (x, q)
dequeue [] = error “empty queue”
```

```
type Queue a = ([a], [a])
([], [])
```

```
type Queue a = ([a], [a])
([1], [])
```

```
type Queue a = ([a], [a])
([2, 1], [])
```

```
type Queue a = ([a], [a])
([3, 2, 1], [])
```

```
type Queue a = ([a], [a])
([], [1, 2, 3])
```

```
type Queue a = ([a], [a])
([], [2, 3])
```

```
type Queue a = ([a], [a])
([4], [2, 3])
```

```
type Queue a = ([a], [a])
([5, 4], [2, 3])
```

```
type Queue a = ([a], [a])
([5, 4], [2, 3])

enqueue :: a → Queue a → Queue a
enqueue x (front, back) = (x:front, back)

dequeue :: Queue a → (a, Queue a)
dequeue ([], []) = error "empty queue"
dequeue (xs, []) = dequeue ([], foldl (:) [] xs))
dequeue (xs, (y:ys)) = (y, (xs, ys))
```

```
(define queue (box empty-queue))

(define (enqueue! item)
  (set-box! queue
            (enqueue item (unbox queue))))

(define (dequeue! _)
  (with* ([val*new-q (dequeue (unbox queue))])
    (seq
      (set-box! queue (snd val*new-q))
      (fst val*new-q))))
```

```
(define (start thunk)
  (enqueue!
    (fun (_)
      (seq (thunk _)
           (finish _)))))
```

```
(define (yield _)
  (let/cc k
    (seq (enqueue! k)
         (with* ([next (dequeue! _)])
           (next _)))))
```

```
(define (finish _)
  (if (empty? (unbox queue))
      _
      (with* ([next (dequeue! _)])
        (next _))))
```

```
type State = (…, Queue Cont, TickCount)
interp :: Expr → Env → State → Cont → Result (Val, State)
```

```
type State = (…, Queue Cont, [Bool])
interp :: Expr → Env → State → Cont → Result (Val, State)
```

```
interpK :: Expr → Env → (Val → Result a) → Result a
interpK expr env k = case expr of
  NumE n → k (NumV n)
  FunE var body → k (FunV var body env)
  IfE cond cons alt →
    interp cond env (\v →
      case v of
        BoolV True → interp cons env k
        BoolV False → interp alt env k
        nonBool → Err ((show nonBool) ++ “: not a boolean”))
```

```
interpK :: Expr → Env → (Val → Result a) → Result a
interpK expr env k = case expr of
  LetCcE var body →
    interp body ((var, ContV k):env) k
  AppE fun arg →
    interp fun env (\fv →
      interp arg env (\av →
        case fv of
          FunV var body closEnv →
            interp body ((var, av):closEnv) k
          PrimV fn → fn av k
          ContV k′ → k′ av
          nonFun → Err ((show nonFun) ++ ": not a function")))
```

```
data Cont = DoneK
          | IfK Expr Expr Env Cont
          | AppFunK Expr Env Cont
          | AppArgK Val Cont
```

```
interpK :: Expr → Env → Cont → Result Val
interpK expr env k = case expr of
  NumE n → callK k (NumV n)
  FunE var body → callK k (FunV var body env)
  IfE cond cons alt →
    interp cond env (IfK cons alt k)
  LetCcE var body →
    interp body ((var, ContV k):env) k
  AppE fun arg →
    interp fun env (AppFunK arg env k)
```

```
data Expr = NumE Int | BoolE Bool | IfE Expr Expr Expr
          | BinOpE Op Expr Expr
```

```
data Val = NumV Int | BoolV Bool
```

```
interp :: Expr → Result Val
```

```
data Expr = NumE Int | BoolE Bool | IfE Expr Expr Expr
          | FunE Var Expr | VarE Var | AppE Expr Expr
type Env = Var → Val



data Val = NumV Int | BoolV Bool
         | FunV (Val → Result Val)


interp :: Expr → Env → Result Val
```

```
data Expr = NumE Int | BoolE Bool | IfE Expr Expr Expr
          | FunE Var Expr | VarE Var | AppE Expr Expr
type Env = Var → Val
type Store = Loc → Val
type STR a = Store → (Result a, Store)

data Val = NumV Int | BoolV Bool
         | FunV (Val → STR Val)
         | BoxV Loc

interp :: Expr → Env → STR Val
```

```
data Expr = NumE Int | BoolE Bool | IfE Expr Expr Expr
          | FunE Var Expr | VarE Var | AppE Expr Expr
type Env = Var → Val
type CPS a = (a → Result Val) → Result Val


data Val = NumV Int | BoolV Bool
         | FunV (Val → CPS Val)
         | ContV (Val → Result Val)

interp :: Expr → Env → CPS Val
```

```
data Expr = NumE Int | BoolE Bool | IfE Expr Expr Expr
          | FunE Var Expr | VarE Var | AppE Expr Expr
type Env = Var → Val
type CPS a = (a → Result Val) → Result Val


data Val = NumV Int | BoolV Bool
         | FunV (Val → CPS Val)


interp :: Expr → Env → CPS Val
```

```
data Expr = NumE Int | BoolE Bool | IfE Expr Expr Expr
          | FunE Var Expr | VarE Var | AppE Expr Expr
type Env = Var → Val
type Store = Loc → Val
type STCPS a = Store → ((a, Store) → (Result Val, Store)) →
                         (Result Val, Store))
data Val = NumV Int | BoolV Bool
         | FunV (Val → STCPS Val)
         | BoxV Loc

interp :: Expr → Env → STCPS Val
```

```
data Expr = NumE Int | BoolE Bool | IfE Expr Expr Expr
          | FunE Var Expr | VarE Var | AppE Expr Expr
type Env = Var → Val



data Val = NumV Int | BoolV Bool
         | FunV (Val → Result Val)


interp :: Expr → Env → Result Val
```

```
data Expr = NumE Int | BoolE Bool | IfE Expr Expr Expr
          | FunE Var Expr | VarE Var | AppE Expr Expr
type Env = [(Var, Val)]
```

```
data Val = NumV Int | BoolV Bool
         | FunV Var Expr Env
```

```
interp :: Expr → Env → Result Val
```

```
data Expr = NumE Int | BoolE Bool | IfE Expr Expr Expr
          | FunE Var Expr | VarE Var | AppE Expr Expr
type Env = [(Var, Val)]




data Val = NumV Int | BoolV Bool
         | FunV Var Expr Env | PrimOpV Id
interpOp :: Id → Val → Result Val

interp :: Expr → Env → Result Val
```

```
data Expr = ValE Val                    | IfE Expr Expr Expr
                          | VarE Var | AppE Expr Expr



data Val = NumV Int | BoolV Bool
         | FunV Var Expr      | PrimOpV Id
interpOp :: Id → Val → Result Val

interp :: Expr → Result Val
```

v ::= n

  | **true** | **false**

  | *op*

  | (**fun** (*x*) e)

e ::= v

  | (**if** e e e)

  | *x*

  | (e e)

$$
\begin{aligned}
v &::= n \\
&\mid \mathbf{true} \mid \mathbf{false} \\
&\mid op \\
&\mid \boldsymbol{\lambda} x.\mathrm{e} \\
\mathrm{e} &::= v \\
&\mid \mathbf{if}\ \mathrm{e}\ \mathrm{e}\ \mathrm{e} \\
&\mid x \\
&\mid \mathrm{e}\ \mathrm{e}
\end{aligned}
$$

v ::= n | **true** | **false** | *op* | **λ***x*.e   e ::= v | **if** e e e | *x* | e e

$$v \Downarrow v \tag{VAL}$$

$$\frac{e_c \Downarrow \text{true} \qquad e_t \Downarrow v}{(\text{if } e_c\ e_t\ e_f) \Downarrow v} \tag{IF-TRUE}$$

$$\frac{e_c \Downarrow \text{false} \qquad e_f \Downarrow v}{(\text{if } e_c\ e_t\ e_f) \Downarrow v} \tag{IF-FALSE}$$

$$\frac{e_f \Downarrow op \qquad e_a \Downarrow v_a \qquad \delta(op, v_a) = v}{(e_f\ e_a) \Downarrow v} \tag{δ}$$

$$\frac{e_f \Downarrow (\lambda x.e_b) \qquad e_a \Downarrow v_a \qquad e_b[x \leftarrow v_a] \Downarrow v}{(e_f\ e_a) \Downarrow v} \tag{β_v}$$

$[x \to s]b = \ldots$

$$[x \to s]x = s$$

$$[x \to s]y = y \qquad (y \neq x)$$

$$[x \to s]\lambda x.b = \lambda x.b$$

$$[x \to s]\lambda y.b = \lambda y.[x \to s]b \quad (y \neq x)$$

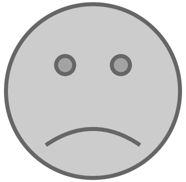
$$[x \to s](e_1\ e_2) = ([x \to s]e_1\ [x \to s]e_2)$$

$$[x \to s](\textbf{if}\ e_1\ e_2\ e_3) = (\textbf{if}\ [x \to s]e_1\ [x \to s]e_2\ [x \to s]e_3)$$

$[x \longrightarrow (f\ (g\ y))]\boldsymbol{\lambda} y.(\textbf{if}\ x\ 3\ y)$

$$[x \longrightarrow (f\ (g\ y))]\boldsymbol{\lambda} y.(\textbf{if}\ x\ 3\ y)$$

$$\Rightarrow \boldsymbol{\lambda} y.(\textbf{if}\ (f\ (g\ y))\ 3\ y)$$



$FV(e) = \ldots$

$FV(x) = \{ x \}$

$FV(e_1\ e_2) = FV(e_1) \cup FV(e_2)$

$FV(\textbf{if}\ e_1\ e_2\ e_3) = FV(e_1) \cup FV(e_2) \cup FV(e_3)$

$FV(\boldsymbol{\lambda} x.e) = FV(e) \setminus \{ x \}$

$FV(v) = \{\}$ (v not of the form $\boldsymbol{\lambda} x.e$)

$[x \longrightarrow (f\ (g\ y))]\boldsymbol{\lambda} y.(\textbf{if}\ x\ 3\ y)$

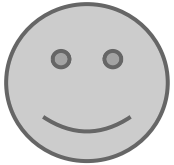
$$[x \to (f\ (g\ y))]\boldsymbol{\lambda} y.(\mathbf{if}\ x\ 3\ y)$$

$$\Rightarrow [x \to (f\ (g\ y))]\boldsymbol{\lambda} z.(\mathbf{if}\ x\ 3\ z)$$

$$[x \mapsto (f\ (g\ y))]\boldsymbol{\lambda} y.(\textbf{if}\ x\ 3\ y)$$

$$\Rightarrow [x \mapsto (f\ (g\ y))]\boldsymbol{\lambda} z.(\textbf{if}\ x\ 3\ z)$$

$$\Rightarrow [x \mapsto (f\ (g\ y))]\boldsymbol{\lambda} z.(\textbf{if}\ (f\ (g\ y))\ 3\ z)$$



$[x \to s]b = \ldots$

$[x \to s]x = s$

$[x \to s]y = y \qquad (y \neq x)$

$[x \to s]\boldsymbol{\lambda} x.b = \boldsymbol{\lambda} x.b$

$[x \to s]\boldsymbol{\lambda} y.b = \boldsymbol{\lambda} y.[x \to s]b \quad (y \neq x \text{ and } y \notin FV(e))$

$[x \to s](e_1\ e_2) = ([x \to s]e_1\ [x \to s]e_2)$

$[x \to s](\textbf{if}\ e_1\ e_2\ e_3) = (\textbf{if}\ [x \to s]e_1\ [x \to s]e_2\ [x \to s]e_3)$

v ::= n | **true** | **false** | *op* | **λ***x*.e   e ::= v | **if** e e e | *x* | e e

$$\dfrac{(\lambda x.x\ x) \Downarrow (\lambda x.x\ x) \qquad (\lambda x.x\ x) \Downarrow (\lambda x.x\ x) \qquad \dfrac{\vdots}{((\lambda x.x\ x)\ (\lambda x.x\ x)) \Downarrow \ldots}}{((\lambda x.x\ x)\ (\lambda x.x\ x)) \Downarrow \ldots}$$

$$\dfrac{\dfrac{(\lambda x.\mathbf{if}\ \ldots) \Downarrow \cdot \qquad 3 \Downarrow 3 \qquad \dfrac{\dfrac{\delta(\text{iszero},3)=\text{false}}{\text{iszero } 3 \Downarrow \text{false}} \qquad \dfrac{\delta(\text{add},3)\Downarrow(3+)}{\text{add } 3 \Downarrow (3+)}}{\mathbf{if}\ (\text{iszero } 3)\ \text{succ}\ (\text{add } 3) \Downarrow (3+)}}{((\lambda x.\mathbf{if}\ \ldots)\ 3) \Downarrow (3+)} \qquad 4 \Downarrow 4 \qquad \delta((3+),4)=7}{(((\lambda x.\mathbf{if}\ (\text{iszero}\ x)\ \text{succ}\ (\text{add}\ x))\ 3)\ 4) \Downarrow 7}$$